Homework 4: due by 11:59pm on Wednesday, April 10, 2019

Instructors: Mackale Joyner, Zoran Budimlić

Checkpoint 1 due by 11:59pm on Wednesday, April 3, 2019
Total score: 100 points

All homeworks should be submitted through the Autograder, and also committed in the svn repository at `https://svn.rice.edu/r/comp322/turnin/S19/`*your-netid*`/hw_4` that we will create for you. In case of problems committing your files, please contact the teaching staff at `comp322-staff@mailman.rice.edu` before the deadline to get help resolving for your issues.

Your solution to the written assignment should be submitted as a PDF file named `hw_4_written.pdf` in the hw_4 directory. This is important — you will be penalized 10 points if you place the file in some other folder or with some other name. The PDF file can be created however you choose. If you scan handwritten text, make sure that you use a proper scanner (not a digital camera) to create the PDF file. Your solution to the programming assignment should be submitted in the appropriate location in the hw_4 directory.

The slip day policy for COMP 322 is similar to that of COMP 321. All students will be given 3 slip days to use throughout the semester. When you use a slip day, you will receive up to 24 additional hours to complete the assignment. You may use these slip days in any way you see fit (3 days on one assignment, 1 day each on 3 assignments, etc.). Slip days will be automatically tracked through the Autograder.

Other than slip days, no extensions will be given unless there are exceptional circumstances (such as severe sickness, not because you have too much other work). Such extensions must be requested and approved by the instructor (via e-mail, phone, or in person) before the due date for the assignment. Last minute requests are likely to be denied.

If you see an ambiguity or inconsistency in a question, please seek a clarification on Piazza (remember not to share homework solutions in public posts) or from the teaching staff. If it is not resolved through those channels, you should state the ambiguity/inconsistency that you see, as well as any assumptions that you make to resolve it.

Finally, as with Homework 3, it is important for you to start early, and to meet the intermediate checkpoint to ensure that you are on track to complete the entire homework before the final deadline.

Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates, the teaching assistants and the professor, but you should never misrepresent someone else's work as your own. If you use any material from external sources, you must provide proper attribution.

1 Written Assignments (40 points total)

Submit your solutions to the written assignments as a PDF file named ***hw_4_written.pdf*** *in the hw_4 directory. Please note that you be penalized 10 points if you misplace the file in some other folder or if you submit the report in some other format.*

1.1 HJ isolated constructs vs. Java atomic variables (20 points)

Many applications use Pseudo-Random Number Generators (PRNGs) as in class IsolatedPRNG in Listing 1. The idea is that the seed field takes a linear sequence of values obtained by successive calls to the nextInt() method as shown in line 6. The use of the HJ isolated construct in lines 4–9 ensures that there will be no data race on the seed field if nextSeed() is called in parallel by multiple tasks. The serialization of the isolated construct instances will determine which task obtains which seed in the sequence.

```
1  class IsolatedPRNG {
2    private int seed;
3    public int nextSeed() {
4      final int retVal = isolatedWithReturn(() -> {
5        final int curSeed = seed;
6        final int newSeed = nextInt(curSeed);
7        seed = newSeed;
8        return curSeed;
9      });
10     return retVal;
11   } // nextSeed()
12   . . . // Definition of nextInt(), constructors, etc.
13 } // IsolatedPRNG
```

Listing 1: Concurrent PRNG implemented using isolated construct

Since the isolated construct is not available in standard Java, class AtomicPRNG in Listing 2 attempts to implement the same functionality by using Java atomic variables instead.

1. (10 points) Assuming a scenario where nextSeed() is called by multiple tasks in parallel on the same PRNG object, state if the implementation of AtomicPRNG.nextSeed() has the same semantics as that of IsolatedPRNG.nextSeed(). If so, why? If not, why not?

   By "same semantics", we mean that for every IsolatedPRNG execution, we can find an equivalent AtomicPRNG execution that results in the same answer, and for every AtomicPRNG execution, we can find an equivalent IsolatedPRNG execution that results in the same answer. For example, it should not be the case that the AtomicPRNG execution skips a seed value that is returned by IsolatedPRNG, or vice versa.

2. (10 points) Why is the "while (true)" loop needed in line 5 of AtomicPRNG.nextSeed()? What would happen if the while(true) loop was replaced by a loop that executes for only one iteration?

```
1  class AtomicPRNG {
2    private AtomicInteger seed;
3    public int nextSeed() {
4      int retVal;
5      while (true) {
6        retVal = seed.get();
7        int nextSeedVal = nextInt(retVal);
8        if (seed.compareAndSet(retVal, nextSeedVal)) break;
9      } // while
10     return retVal
11   } // nextSeed()
12   . . . // Definition of nextInt(), constructors, etc.
13 } // AtomicPRNG
```

Listing 2: Concurrent PRNG implemented using Java's AtomicInteger class

1.2 Written Assignment: Dining Philosophers Problem (20 points)

In Lecture 27, we studied the characteristics of different solutions to the Dining Philosophers problem. Both Solution 1 (using Java's synchronized statement) and Solution 2 (using Java's lock library) had the following structure in which each philosopher attempts to first acquire the left fork, and then the right fork:

```
final int numPhilosophers = 5;
final int numForks = numPhilosophers;
final Fork[] fork = ... ; // Initialize array of forks
forall(0, numPhilosophers-1, (p) -> {
  while(true) {
    Think ;
    Acquire left fork, fork[p] ;
    Acquire right fork, fork[(p-1)%numForks] ;
    Eat ;
  } // while
}); // forall
```

Consider a variant of this approach in which any 4 philosophers follow the above structure, but the 1 remaining philosopher first attempts to acquire the right fork and then the left fork.

1. (10 points) Is a deadlock possible in Solution 1 (using Java's synchronized statement) for this variant with 4 philosophers attempting to first acquire the left fork, and then the right fork, and the fifth philosopher doing the opposite? If so, show an execution that leads to deadlock. If not, explain why not.

2. (10 points) Is a livelock possible in Solution 2 (using Java's lock library) for this variant with 4 philosophers attempting to first acquire the left fork, and then the right fork, and the fifth philosopher doing the opposite? If so, show an execution that exhibits a livelock. If not, explain why not. For the purpose of this program, a livelock is a scenario in which all philosophers starve without any of them being blocked.

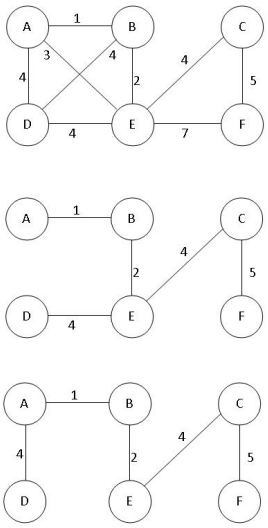

Figure 1: Two possible Minimum Spanning Trees (MSTs) for the undirected graph shown at the top (source: `http://en.wikipedia.org/wiki/Minimum_spanning_tree`)

2 Programming Assignment: Minimum Spanning Tree of an Undirected Graph (60 points)

In this homework, we will focus on the problem of finding a *minimum spanning tree* of an undirected graph. Some of you may recall minimum spanning trees from COMP 182. Note that we have studied parallel spanning tree algorithms earlier in COMP 322, but the focus of this assignment is on parallel algorithms for finding the spanning tree with minimum cost.

The following definition is from Wikipedia (`http://en.wikipedia.org/wiki/Minimum_spanning_tree`):

> Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A Minimum Spanning Tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

Figure 1 shows there may be more than one minimum spanning tree in a graph. In the figure, two minimum spanning trees are shown for the given graph (each with total cost = 16).

2.1 Boruvka's algorithm to compute the Minimum Spanning Tree of an Undirected Graph

The first known algorithm for finding the MST of an undirected graph was developed by Czech scientist Otakar Boruvka in 1926. Two other commonly used algorithms for computing the MST are Prim's algorithm and Kruskal's algorithm. In this assignment, we will focus on parallelizing Boruvka's algorithm, by providing a reference sequential implementation of that algorithm in Java. *If you prefer to parallelize an alternate MST algorithm, please send email to comp322-staff@rice.edu as soon as possible. You will then be responsible for obtaining or creating a sequential Java implementation of that algorithm as a starting point.*

The following summary of Boruvka's sequential algorithm is from the UT Austin Galois project's description

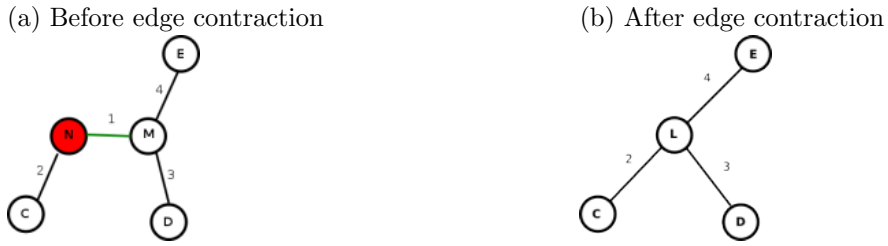

Figure 2: Demonstration of edge contraction

of Boruvka's algorithm:

> Boruvka's algorithm computes the minimal spanning tree through successive applications of edge-contraction on an input graph (without self-loops). In edge-contraction, an edge is chosen from the graph and a new node is formed with the union of the connectivity of the incident nodes of the chosen edge. In the case that there are duplicate edges, only the one with least weight is carried through in the union. Figure 2 demonstrates this process. Boruvka's algorithm proceeds in an unordered fashion. Each node performs edge contraction with its lightest neighbor.

In the example in Figure 2, the edge connecting nodes M and N is contracted, resulting in the replacement of nodes M and N by a single node, L.

The Maven project for this lab is located in the following svn repository:

- `https://svn.rice.edu/r/comp322/turnin/S19/`*`NETID`*`/hw_4`

Please use the subversion command-line client to checkout the project into appropriate directories locally.

2.2 Reference Sequential Implementation of Boruvka's algorithm

The files `SeqBoruvka.java`, `SeqComponent.java`, `SeqEdge.java` contain a reference sequential implementation of Boruvka's algorithm in Java.

```
1  // START OF EDGE CONTRACTION ALGORITHM
2  Component n = null;
3  while (!nodesLoaded.isEmpty()) {
4    // poll() removes first element from the nodesLoaded work-list
5    n = nodesLoaded.poll();
6    if (n.isDead)
7      continue; // node n has already been merged
8    Edge e = n.getMinEdge(); // retrieve n's edge with minimum cost
9    if (e == null)
10     break; // done - we've contracted the graph to a single node
11   Component other = e.getOther(n);
12   // mark this component dead, i.e. we have visited this component
13   other.isDead = true;
14   n.merge(other, e.weight); // Merge node other into node n
15   nodesLoaded.add(n); // Add newly merged n back in the work-list
16 }
17 // END OF EDGE CONTRACTION ALGORITHM
```

Listing 3: Sequential version of Boruvka's MST algorithm

Listing 3 shows the main code for Boruvka's algorithm from `SeqBoruvka.java`. The field, `nodesLoaded`, is a reference to a *work-list* that (in the sequential version) is implemented as a `LinkedList` of `Component` objects, where each component refers to a collection of one or more more nodes that have been contracted. Initially, each node is in a component by itself. This implementation assumes that the graph contains no self-loops (*i.e.,* no edge from a node to itself) and that the graph is connected.

Each iteration of the `while` loop in line 5 removes a component, `n`, from the work-list[1]. Line 6 skips the component if it was marked as *dead i.e.,* if it was merged with another component. Line 8 retrieves edge `e` with minimum cost connected to component `n`. If `n` has no adjacent edges, then we must have collapsed all the nodes into a single component and we can exit the loop using the `break` statement in line 10. Line 11 sets `other` to the component connected to `n` at the other end of `e`. Lines 13 and 14 do the necessary book-keeping to merge `other` into `n`. Finally, line 15 adds the newly contracted component, `n`, back into the work-list.

2.3 Roadway Data

There are several `.gz` files under `src/main/resources/boruvka` for evaluating the correctness and performance of your parallel Boruvka implementation. These files contain distance graphs that represent the roadways of various regions in the United States (`http://www.dis.uniroma1.it/challenge9/download.shtml`). Each edge in these graphs represents the distance between two locations on a road. For example, `Boruvka USA-road-d.NY.gr.gz` contains the distance graph for the state of New York.

2.4 Your Assignment: Parallel Minimum Spanning Tree Algorithm

Your assignment is to design and implement a parallel algorithm for finding the minimum spanning tree of an undirected graph using whatever parallel Java primitives you have learned in this class — standard Java threads, `java.util.concurrent` libraries, HJ library, or some (carefully selected) combination thereof. You can use the sequential implementation of Boruvka's algorithm described in Section 2.2 as a starting point (in the provided `ParBoruvka.java` file), and you are free to modify any files in the `edu.rice.comp322.boruvka.parallel` package that you choose.

There are two main items to note in the provided code:

1. The `runIteration` method in `ParBoruvka` accepts a number of threads argument. This argument will reflect the number of hardware cores allocated to your experiment when testing your solution after submission and through the autograder. You are free to ignore this parameter, but it can help you to tune your solution by not creating too many threads for the given number of cores.
2. The `ParBoruvka` class includes a `usesHjLib` method that defaults to true. This method indicates to the tests whether your implementation requires that the HJlib runtime be created before starting your experiments. This method defaults to true, indicating that you are using HJlib. If you are not using HJlib, this should be changed to false to ensure the HJlib runtime is not initialized to prevent HJlib threads from conflicting with any Java threads manually created by you.

Your homework will be evaluated as follows. Though your Checkpoint 1 submission will be graded separately from the final submission, it is important that your final submission also pass Checkpoint 1.

1. **[Checkpoint 1 due on April 3, 2019: correct parallel algorithm (20 points)]**

   The following two observations about the algorithm in Listing 3 can provide insights on how to parallelize the algorithm:

   - The order in which components are removed (line 5) from and inserted (line 15) in the work-list is not significant i.e., the work-list can be implemented as any unordered collection.
   - If two iterations of the `while` loop work on disjoint (`n`, `other`) pairs, then the iterations can be executed in parallel using a thread-safe implementation of the work-list that allows inserts and removes to be invoked in parallel.

[1]Line numbers here refer to Listing 3, and not the code in `SeqBoruvka.java`.

You will get full credit for this checkpoint with any correct implementation that passes all the unit tests while exploiting parallelism among while-loop iterations as indicated above, even if the implementation incurs large overheads and runs slower than the sequential version. Your program should return a correct MST solution for the given test input, when executed with at least 2 threads. Include the output from one run of the provided correctness tests in `BoruvkaCorrectnessTest.java` in your report.

2. [**Performance evaluation on NOTS compute nodes (25 points)**] The goal of this part of the homework is to evaluate the scalability of your parallel implementation. Measure the performance of your program when using 1, 2, 4, 6, and 8 threads, and compare it with the performance of the provided sequential version when processing the `USA-road-d.NE.gr.gz` graph (using the autograder will automatically run these scaling tests for you using the provided `BoruvkaPerformanceTest.testInputUSAroadNE` test).

   You will be graded on the performance obtained by your parallel implementation with 2, 4, 6, and 8 threads, relative to the performance of the same parallel implementation using 1 thread (note that this is scalability, not speedup). Getting good speedups when parallelizing Boruvka's algorithm can be challenging. However, you should see some performance improvements when going from 1 to 2 or 4 threads. Do not be surprised if you see performance degradations with 6 or 8 threads.

   The 25 points for performance evaluation will be broken down as follows:

   1-thread execution time — 5 points if the 1-thread execution time is $\leq 2\times$ the sequential time, 1 point if it terminates correctly (regardless of performance), and 0 points if the 1-thread execution does not terminate with the correct answer.

   2-thread execution time — 5 points if the 2-thread execution time is $\leq 2/3\times$ the 1-thread time (speedup ≥ 1.5), 2 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 1 point if it terminates correctly (regardless of performance), and 0 points if the 2-thread execution does not terminate with the correct answer.

   4-thread execution time — 5 points if the 4-thread execution time is $\leq 4/7\times$ the 1-thread time (speedup ≥ 1.75), 2 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 1 point if it terminates correctly (regardless of performance), and 0 points if the 4-thread execution does not terminate with the correct answer.

   6-thread execution time — 5 points if the 6-thread execution time shows a speedup ≥ 1.9, 2 points if it is $\leq$ the 1-thread time (speedup ≥ 1), 1 point if it terminates correctly (regardless of performance), and 0 points if the 6-thread execution does not terminate with the correct answer.

   8-thread execution time — 5 points if the 8-thread execution time is $\leq 1/2\times$ the 1-thread time (speedup ≥ 2), 1 point if it terminates correctly (regardless of performance), and 0 points if the 8-thread execution does not terminate with the correct answer.

3. [**Homework report (15 points)**]

   A report file formatted as a PDF file named `hw_4_report.pdf` in the hw_4 directory. The report should contain the following:

   (a) A summary of the design and implementation of the parallel constructs used in your parallel MST algorithm.

   (b) An explanation as to why you believe that your implementation is correct and data-race-free.

   (c) An explanation as to why you believe that your implementation will never result in a deadlock.

   (d) An explanation as to why you believe that your implementation will never result in a livelock.

   Your report should also include the following measurements for both parts 1 and 2:

   (a) Output of the provided correctness tests in `BoruvkaCorrectnessTest.java`.

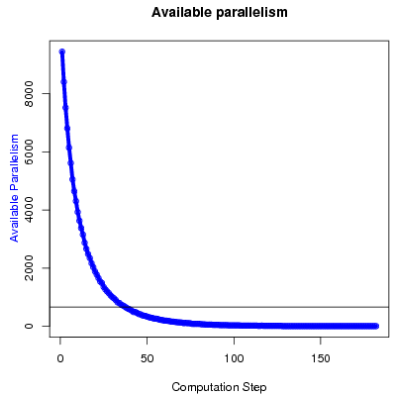

Figure 3: Available parallelism in parallel Boruvka Minimum Spanning Tree algorithm (source: Galois MST description).

(b) Output of the provided performance tests in `BoruvkaPerformanceTest.java` for your selected input, executed with 1, 2, 4, 6, and 8 threads.

Please place the report file(s) in the top-level `hw_4` directory. Remember to commit all your source code into the subversion repository during your assignment submission.

2.5 Some Implementation Tips

Here are some tips to keep in mind for this homework:

- Given the large overhead of creating Java threads, you should try and get by with just creating a small number of threads (*e.g.,* one thread per processor core), or a small number of workers if you use HJlib. Each thread can then share the work of the while-loop in Listing 3.
- Figure 3 illustrates how the available parallelism decreases as more and more merge steps are performed. This suggests that the actual parallelism exploited (e.g., number of threads) should be reduced as the contracted graph reaches certain threshold sizes (perhaps even going down to sequential execution at the very end).
- The work-list is implemented as a `LinkedList` in the sequential version of `SeqBoruvka.java`. It will need to be implemented as a thread-safe collection when multiple `while` iterations are executed in parallel. Two `java.util.concurrent` classes that can be useful for this purpose are `ConcurrentLinkedQueue` and `ConcurrentHashMap`. You're welcome to implement your own data structure if you choose *e.g.,* by using `AtomicInteger` operations to manage indexing in a shared array. (Note that the number of insertions in the work-list is bounded by the initial number of nodes.)
- In addition to changes in `ParBoruvka.java`, you may find it convenient to modify `ParComponent.java` to add some form of mutual exclusion constructs to manage cases when two threads collide on the same node when trying to contract a pair. For mutual exclusion, you can try using Java's built-in locks with the `synchronized` statement and the use of `wait-notify` operations as needed, or you can explicitly allocate a `java.util.concurrent.locks.ReentrantLock` for each node/component. One advantage of `ReentrantLock` is that it supports a `tryLock()` method that allows a thread to query the status of a lock without blocking on the request. The potential disadvantage is that it incurs extra space overhead, which may indirectly impact execution time.